CS370 Operating Systems
Midterm Review

Yashwant K Malaiya
Fall 2022
Safe State, Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

System is in safe state if there exists a sequence \(<P_1, P_2, ..., P_n>\) of ALL the processes such that

• for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by
  – currently available resources +
  – resources held by all the \(P_j\), with \(j < i\)
  – That is
    • If \(P_i\) resource needs are not immediately available, then \(P_i\) can wait until all \(P_j\) have finished and released resources
    • When \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, and so on

• If no such sequence exists: system state is unsafe
Example A: Assume 12 Units in the system

<table>
<thead>
<tr>
<th></th>
<th>Max need</th>
<th>Current holding</th>
</tr>
</thead>
<tbody>
<tr>
<td>av</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

At time T0 (shown):
9 units allocated
3 (12-9) units available

A unit could be a drive, a block of memory etc.

- Is the system at time T0 in a safe state?
  - Try sequence <P1, P0, P2>
  - P1 can be given 2 units
  - When P1 releases its resources; there are now 5 available units
  - P0 uses 5 and subsequently releases them (10 available now)
  - P2 can then proceed.

- Thus <P1, P0, P2> is a safe sequence, and at T0 system was in a safe state
Example A: Assume 12 Units in the system (timing)

<table>
<thead>
<tr>
<th>Max need</th>
<th>Current holding</th>
<th>+2 allo to P1</th>
<th>P1 releases all</th>
<th>..</th>
<th>..</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
</tr>
<tr>
<td>av</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10 done</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>4 done</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Thus the state at T0 is safe.
Example B: 12 Units initially available in the system

<table>
<thead>
<tr>
<th>Max need</th>
<th>T0</th>
<th>T1 safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

Before T1: 3 units available
At T1: 2 units available

- At time T1, P2 is allocated 1 more units. Is that a good decision?
  - Now only P1 can proceed (already has 2, and given be given 2 more)
  - When P1 releases its resources; there are 4 units
  - P0 needs 5 more, P2 needs 6 more. Deadlock.
    - **Mistake** in granting P2 the additional unit.
- The state at T1 is not a safe state. Wasn’t a good decision.
Review for Midterm


- Sec 001 and local 801 students
  - 2-3:15 PM Tuesday Oct 10 in Biology 136 usual room
- Sec 801 (non-local):
  - 1 hr 15 min. Tuesday Oct 10 2 PM- Wed March 9 3:15 PM window.

- SDC students: You should have made arrangements with SDC already.
- Sec 001: Respondus lockdown browser, Sec 801: Honorlock
How to prepare for the Midterm

What you have been doing already

• Attend classes, listen actively, review slides
• Quizzes: Review things before and during quizzes spending more time is better
• Self Exercises and Homework: Understand objectives & constructs, design approach, review & test code
• Study before exams. Why?
Computer System Structures

- Computer System Operation
  - Stack for calling functions (subroutines)
- I/O Structure: polling, interrupts, DMA
- Storage Structure
  - Storage Hierarchy
- System Calls and System Programs
- Command Interpreter
The Concept of a Process

- Process - a program in execution
  - process execution proceeds in a sequential fashion
- Multiprogramming: several programs apparently executing “concurrently”.
- Process States
  - e.g., new, running, ready, waiting, terminated.

![Diagram of process states and transitions](image-url)
CPU Switch From Process to Process

C structure

(task_struct)

struct task_struct
process information

struct task_struct
process information

...
Process Creation

- Processes are created and deleted dynamically
- Process which creates another process is called a parent process; the created process is called a child process.
- Result is a tree of processes
  - e.g. UNIX - processes have dependencies and form a hierarchy.
- Resources required when creating process
  - CPU time, files, memory, I/O devices etc.

```plaintext
init
  pid = 1

login
  pid = 8415

kthreadd
  pid = 2

sshd
  pid = 3028

bash
  pid = 8416

khelper
  pid = 6

pdflush
  pid = 200

sshd
  pid = 3610

ps
  pid = 9298

emacs
  pid = 9204

tcsch
  pid = 4005
```

```
cid = fork();
if (cid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed\n");
    return 1;
}
else if (cid == 0) { /* child process */
    execp("/bin/ls", "ls", NULL);
}
else { /* parent process, will wait for child to complete */
    wait(NULL);
}
```
Threads

• A thread (or lightweight process)
  • basic unit of CPU utilization; it consists of:
    – program counter, register set and stack space
  – A thread shares the following with peer threads:
    – code section, data section and OS resources (open files, signals)
  – Collectively called a task.

• Thread support in modern systems
  – User threads vs. kernel threads, lightweight processes
  – 1-1, many-1 and many-many mapping

• Implicit Threading (e.g. OpenMP)
• Hardware support in newer processors
Producer-Consumer Problem

• Paradigm for cooperating processes;
  – producer process produces information that is consumed by a consumer process.

• We need buffer of items that can be filled by producer and emptied by consumer.
  – Unbounded-buffer
  – Bounded-buffer

• Producer and Consumer must synchronize.

```c
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```
Interprocess Communication (IPC)

• Mechanism for processes to communicate and synchronize their actions.
  • Via shared memory
  • Pipes
  • Sockets
  • Via Messaging system - processes communicate without resorting to shared variables.

```c
int fd[2];

create the pipe:
if (pipe(fd) == -1) {
  fprintf(stderr,"Pipe failed");
  return 1;
} else {
  // fd[0] is the read end, fd[1] is the write end
}

fork a child process:
pid = fork();

parent process:
  /* close the unused end of the pipe */
  close(fd[READ_END]);

  /* write to the pipe */
  write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

  /* close the write end of the pipe */
  close(fd[WRITE_END]);

child process:
  ....
```
CPU Scheduling

- **CPU utilization** – keep the CPU as busy as possible: **Maximize**
- **Throughput** – # of processes that complete their execution per time unit: **Maximize**
- **Turnaround time** – time to execute a process from submission to completion: **Minimize**
- **Waiting time** – amount of time a process has been waiting in the ready queue: **Minimize**
- **Response time** – time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): **Minimize**
Scheduling Policies

• FCFS (First Come First Serve)
  – Process that requests the CPU \textit{FIRST} is allocated the CPU \textit{FIRST}.

• SJF (Shortest Job First)
  – Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

• Shortest-remaining-time-first (preemptive SJF)
  – A process preempted by an arriving process with shorter remaining time

• Priority
  – A priority value (integer) is associated with each process. CPU allocated to process with highest priority.

• Round Robin
  – Each process gets a small unit of CPU time

• MultiLevel
  – ready queue partitioned into separate queues
  – Variation: Multilevel Feedback queues: priority lower or raised based on history

• Completely Fair
  – Variable time-slice based on number and priority of the tasks in the queue.
  – virtual run time is the weighted run-time
Example: SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- All arrive at time 0.
- SJF scheduling chart

```
  0 3 9 16 24
P_4 P_1 P_3 P_2
```

- Average waiting time for $P_1, P_2, P_3, P_4 = (3 + 16 + 9 + 0) / 4 = 7
Determining Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using *exponential averaging*
  1. \( t_n \) = actual length of \( n^{th} \) CPU burst
  2. \( \tau_{n+1} \) = predicted value for the next CPU burst
  3. \( \alpha, 0 \leq \alpha \leq 1 \)
  4. Define: \( \tau_{n=1} = \alpha t_n + (1-\alpha)\tau_n \).

- Commonly, \( \alpha \) set to ½
Example of RR with Time Quantum = 4

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Arrive a time 0 in order $P_1$, $P_2$, $P_3$: The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
</tbody>
</table>

- Waiting times: $P_1$:10-4 =6, $P_2$:4, $P_3$:7, average $17/3 = 5.66$ units
- Typically, higher average turnaround than SJF, but better *response*
- $q$ should be large compared to context switch time
- $q$ usually 10ms to 100ms, context switch overhead < 1%

Response time: Arrival to beginning of execution: $P_2$: 4
Turnaround time: Arrival to finish of execution: $P_2$: 7
Multiple-Processor Scheduling

• CPU scheduling more complex when multiple CPUs are available.

• **Assume Homogeneous processors** within a multiprocessor

• **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

• **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
  – all processes in common ready queue, or
  – each has its own private queue of ready processes
    • Currently, most common

• **Processor affinity** – process has affinity for processor on which it is currently running because of info in cache
  – **soft affinity**: try but no guarantee
  – **hard affinity** can specify processor sets
**Consumer-producer problem**

**Producer**

```java
while (true) {
    /* produce an item*/
    while (counter == BUFFER_SIZE) ;
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

**Consumer**

```java
while (true) {
    while (counter == 0);
    /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in
    next consumed */
}
```

They run “concurrently” (or in parallel), and are subject to context switches at unpredictable times.
Race Condition

counter++ could be compiled as

\begin{align*}
\text{register1} &= \text{counter} \\
\text{register1} &= \text{register1} + 1 \\
\text{counter} &= \text{register1}
\end{align*}

counter-- could be compiled as

\begin{align*}
\text{register2} &= \text{counter} \\
\text{register2} &= \text{register2} - 1 \\
\text{counter} &= \text{register2}
\end{align*}

They run concurrently, and are subject to context switches at unpredictable times.

Consider this execution interleaving with “count = 5” initially:

\begin{align*}
\text{S0: producer execute } & \text{register1} = \text{counter} & \{\text{register1} = 5\} \\
\text{S1: producer execute } & \text{register1} = \text{register1} + 1 & \{\text{register1} = 6\} \\
\text{S2: consumer execute } & \text{register2} = \text{counter} & \{\text{register2} = 5\} \\
\text{S3: consumer execute } & \text{register2} = \text{register2} - 1 & \{\text{register2} = 4\} \\
\text{S4: producer execute } & \text{counter} = \text{register1} & \{\text{counter} = 6\} \\
\text{S5: consumer execute } & \text{counter} = \text{register2} & \{\text{counter} = 4\}
\end{align*}

Overwrites!
The Critical Section Problem

- Requirements
  - Mutual Exclusion
  - Progress
  - Bounded Waiting
- Solution to the critical section problem

```plaintext
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```
Peterson’s Algorithm for Process $P_i$

```plaintext
\begin{align*}
\text{do} \quad & \{ \\
\quad & \text{flag}[i] = \text{true}; \\
\quad & \text{turn} = j; \\
\quad & \text{while} \ (\text{flag}[j] \land \text{turn} == j); \\
\quad & \quad \text{critical section} \\
\quad & \text{flag}[i] = \text{false}; \\
\quad & \quad \text{remainder section} \\
\} \quad \text{while} \ (\text{true});
\end{align*}
```

- The variable \text{turn} indicates whose turn it is to enter the critical section
- \text{flag}[i] = \text{true} implies that process $P_i$ is ready!
- Proofs for **Mutual Exclusion, Progress, Bounded Wait**

**Being nice!**
Solution using `test_and_set()`

```c
Solution:
do {
   while (test_and_set(&lock)); // do nothing */
   /* critical section */
   ...
   lock = false;
   /* remainder section */
   ...
} while (true);
```
For process \( i \):

do {
    waiting\[i\] = true;
    key = true;
    while (waiting\[i\] && key)
        key = test_and_set(\&lock);
    waiting\[i\] = false;
    /* critical section */
    j = (i + 1) \% n;
    while ((j != i) && !waiting[j])
        j = (j + 1) \% n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);

Shared Data structures initialized to FALSE

- boolean waiting[n];
- boolean lock;

The entry section for process \( i \):
- First process to execute TestAndSet will find key == false; ENTER critical section,
- EVERYONE else must wait

The exit section for process \( i \):
Part I: Finding a suitable waiting process \( j \) and enable it to get through the while loop,
or if there is no suitable process, make lock FALSE.
Mutex Locks

- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

```c
acquire() {
    while (!available); /* busy wait */
}
```

```c
release() {
    available = true;
}
```

**Usage**

```c
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```
Semaphore

• Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
• Semaphore $S$ – integer variable
• Can only be accessed via two indivisible (atomic) operations
  – *wait()* and *signal()*
    • Originally called *p()* and *v()*
• Definition of the *wait()* operation
  
  ```c
  wait(S) {
      while (S <= 0) // busy wait
          S--;
  }
  ```

• Definition of the *signal()* operation
  
  ```c
  signal(S) {
      S++;
  }
  ```
Wait(S) and Signal (S)

Process 0
- Wait(S)
- Critical section
- Signal (S)

Semaphore S
- S = 1
- S = 0
- S = 1
- Locked by Process 1

Process 1
- Wait (S)
- Busy waiting
- Gets lock, S-
- Critical section
- Signal (S)
Readers-Writers Problem (Cont.)

- The structure of a reader process
  
  ```
  do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
          wait(rw_mutex);
      signal(mutex);
      ...
      /* reading is performed */
      ...
      wait(mutex);
      read_count--;
      if (read_count == 0)
          signal(rw_mutex);
      signal(mutex);
  } while (true);
  ```

- The structure of a writer process
  
  ```
  do {
      wait(rw_mutex);
      ...
      /* writing is performed */
      ...
      signal(rw_mutex);
  } while (true);
  ```

When: writer in critical section and if n readers waiting
1 is queued on rw_mutex (n-1) queued on mutex
Monitors and Condition Variables

The `condition` construct

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` — a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` — resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable. *Signal is lost.*
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);  //on next slide
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
Deadlocks

- **System Model**
  - Resource allocation graph, claim graph (for avoidance)

- **Deadlock Characterization**
  - Conditions for deadlock - mutual exclusion, hold and wait, no preemption, circular wait.

- **Methods for handling deadlocks**
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock

- Combined Approach to Deadlock Handling

At this point, two minimal cycles exist in the system:

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes $P_1$, $P_2$, and $P_3$ are deadlocked.
Deadlock Prevention

– If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.

– Restrain ways in which requests can be made

  • Mutual Exclusion - cannot deny (important)
  • Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.
  • No Preemption
    – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.
  • Circular Wait
    – Impose a total ordering of all resource types.
Deadlock avoidance: Safe states

• If the system can:
  – Allocate resources to each process in some order
    • Up to the maximum for the process
  – Still avoid deadlock
  – Then it is in a **safe state**

• A system is safe ONLY IF there is a safe sequence

• A safe state is not a deadlocked state
  – Deadlocked state is an unsafe state
  – Not all unsafe states are deadlock
Questions

Various types of questions:
• Easy, hard, middle

**Question types** *(may be similar to quiz questions):*
• Problem solving/analyzing: Gantt charts, tables, e.g., scheduling
• True/False, Multiple choice
• Match things
• Identifying things in diagrams or complete them
• Concepts: define/explain/fill in blanks
• Code fragments: fill missing code, values of variables
• How will you achieve something?
• Others
How to prepare for the Midterm

• What you have been doing already
  – Listen to the lectures carefully, connecting terms, concepts and approaches
  – Think while answering quizzes, reviewing material as needed
  – Understanding, designing, coding and testing of programs

• Review course materials
  – Slides
  – HWs
  – Quizzes. There will be one this weekend.
  – Textbook
That’s it for today.