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
Process Concept

- 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.
CPU Switch From Process to Process

**C structure**

```
task_struct
```

- process state
- process number
- program counter
- registers
- memory limits
- list of open files

```
struct task_struct
  process information
  ...
  ...
```

- current (currently executing process)
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.

```c
int pid = 1

sshd pid = 3028
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

fid = 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);
}
```

Colorado State University
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;
}

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]);
```
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 FIRST is allocated the CPU 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

• Other
Example of 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
P4 P1 P3 P2
```

- 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 \( \frac{1}{2} \)
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>0</th>
<th>4</th>
<th>7</th>
<th>10</th>
<th>14</th>
<th>18</th>
<th>22</th>
<th>26</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
This is temporal multithreading. Simultaneous multithreading allows threads to compute in parallel.
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_SIZ
    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

- register1 = counter
- register1 = register1 + 1
- counter = register1

Counter-- could be compiled as

- register2 = counter
- register2 = register2 - 1
- counter = register2

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

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

S0: producer execute register1 = counter
    {register1 = 5}
S1: producer execute register1 = register1 + 1
    {register1 = 6}
S2: consumer execute register2 = counter
    {register2 = 5}
S3: consumer execute register2 = register2 - 1
    {register2 = 4}
S4: producer execute counter = register1
    {counter = 6}
S5: consumer execute counter = register2
    {counter = 4}

Overwrites!
The Critical Section Problem

– Requirements
  – Mutual Exclusion
  – Progress
  – Bounded Waiting

– Solution to the critical section problem

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

```c
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j); /*Wait*/
    critical section
    flag[i] = false;
    remainder section
} while (true);
```

- The variable `turn` indicates whose turn it is to enter the critical section
- `flag[i] = true` implies that process $P_i$ is ready!
- Proofs for Mutual Exclusion, Progress, Bounded Wait
Shared Boolean variable lock, initialized to FALSE

Solution:

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

```c
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

Usage

```c
acquire lock
critical section
release lock
remainder section
```

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

release() {
    available = 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
  ```
  wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
  }
  ```
- Definition of the `signal()` operation
  ```
  signal(S) {
    S++; 
  }
  ```
Wait(S) and Signal (S)

Process 0

Semaphore S

S = 1

Locked by Process 1

S = 0

S = 1

S = 0

S = 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);
  ```

  *mutex for mutual exclusion to readcount*

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

  The structure of a writer process
  
  ```
  do {
      wait(rw_mutex);
      ...
      /* writing is performed */
      ...
      signal(rw_mutex);
  } while (true);
  ```
Implementation with no Busy waiting (Counting Sema)

```c
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

typedef struct{
    int value;
    struct process *list;
} semaphore;

Colorado State University
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.*

```plaintext
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure Pn (...) {……}
    Initialization code (...) { ... }
}
```
The pickup() and putdown() operations

```
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.