Process Synchronization

ECE595

Jan 20

Y. Charlie Hu

Roadmap

- What is a process?
- Concurrent proceses
- Process creation

- Non-preemptive CPU scheduling
- Process synchronization

Multiprogramming needs CPU scheduling

- Without any hardware support, what can the OS do to a running process?

System calls may trigger Scheduler

- Block – wait on some event/resource
  - Network packet arrival (e.g., recv())
  - Keyboard, mouse input (e.g., getchar())
  - Disk activity completion (e.g., read())

- Yield – give up running for now

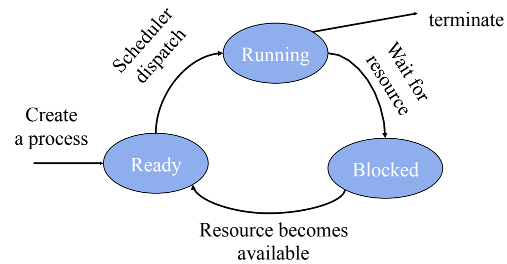
Non-Preemptive Scheduler

- A non-preemptive scheduler: a scheduler that is only invoked by explicit block/yield calls, or terminations
  - Only method when there is no timer!
- The simplest form

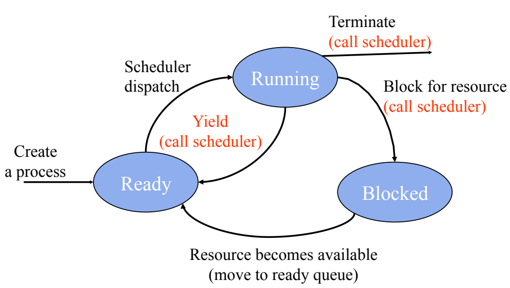
  Scheduler:

  save current process state (into PCB)
  choose next process to run
  dispatch (load PCB and run)
- Used in Windows 3.1, Mac OS

Our Friend -- the Transition Diagram



Process State Transition of Non-Preemptive Scheduling



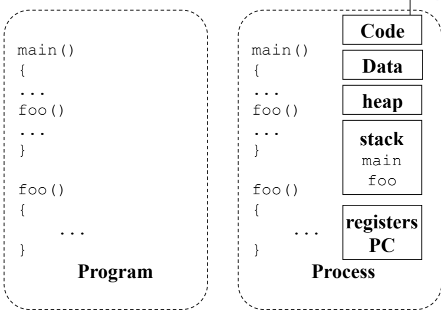
How does a process terminate itself?

Context Switch

- Definition:

  switching the CPU to run another process, which involves (1) saving the state of the old process and (2) loading the state of the new process
- What state?

[lec3] Program vs. Process



Context Switch

- Definition:
  switching the CPU to run another process, which involves (1) saving the state of the old process and (2) loading the state of the new process
- What state?
  - What about L1/L2 cache content?

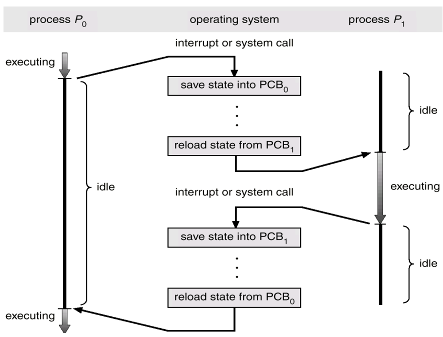
Context Switch

- Definition:
  switching the CPU to another process, which involves saving the state of the old process and load the state of the new process
- What state?
- Where to store them?

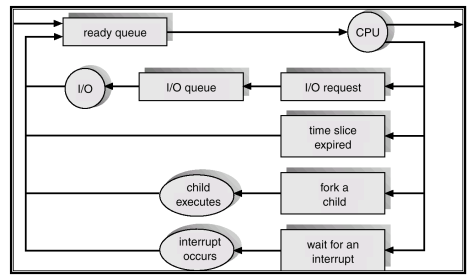
[lec3] Process Control Block (Process Table)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.

Context Switch



Process Scheduling Diagram

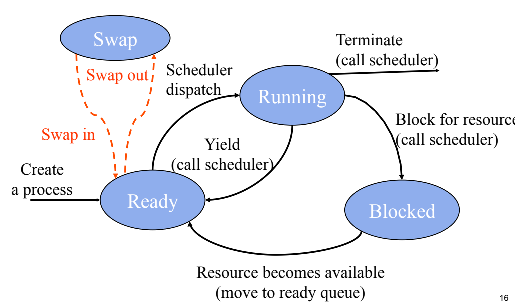


Physical Memory & Multiprogramming

- Want to run many programs
- Programs need memory to run
- Memory is a scarce resource

- What happens when M(a) + M(b) + M(c) > physical mem?

Add Job Swapping to State Transition Diagram



Process synchronization

- Cooperating processes may share data via
  - shared address space (code, data, heap) by using threads
  - shared memory objects (used in lab2)
  - Files
  - (Sending messages)
- What can happen if processes try to access shared data (address) concurrently?
  - Sharing bank account with sibling:

    At 3pm: If (balance > $10) withdraw $10
- How hard is the solution?

"Got Milk?"

- "Too much milk" problem

Process synchronization needs help from OS!

Mutual exclusion & Critical Section

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time
- *Mutual exclusion* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

Example of Critical Section

- Concurrent accesses to shared variables, at least one of which is write

P0:	P1:	P2:
Read note;	Read note;	write note;
…	…	…

Desirable properties of MuEx

- *Fair*: if several processes are waiting, let each in eventually
- *Efficient*: don't use up substantial amounts of resources when waiting (e.g. no busy waiting)
- *Simple*: should be easy to use (e.g. just bracket the critical sections)

Desirable properties of processes using MuEx

- Always lock before manipulating shared data
- Always unlock after manipulating shared data
- Do not lock again if already locked
- Do not unlock if not locked by you
- Do not spend large amounts of time in critical section

Mutual Exclusion provided by OS (and language/compiler)

- Locks
  - Alone can solve simple problems
- More advanced
  - Semaphore
  - Lock and condition variable
    - Lock alone is not flexible enough
  - Monitor
- Each primitive itself is atomic!

Lock (aka mutex)

Init: lock = 1; // 0 means held; 1 means free

lock_acquire(lock)

```
{
   while (lock==0);
   lock--;
}
```

lock_release(lock)

```
{
   if (lock == 0)
      lock++;
}
```

- Each primitive is atomic
- In reality, lock is not implemented as above!
  - The waiting process is put to sleep

"Too much milk" problem with locks

```
Acquire(lock);
if (noMilk)
  buy milk;
Release(lock);
```

} Critical Section

- What is the problem with this solution?

Deep thinking (Homework)

- How can we solve the problem?

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```


Often times, we have to wait for shared resources

Producer & Consumer Problem (1-pool version)

- Producer: creates copies of a resource
- Consumer: uses up (destroys) copies of a resource. (may produce something else)
- Buffer: used to hold resource produced by producer before consumed by consumer
- Synchronization: keeping producer & consumer in sync
- Happens inside OS all the time (e.g. I/Os)
  - How about in real life?

Producer & Consumer – solution using locks?

Producer

```
while (1) {

produce an item;

while (buffer is full);

insert item into buffer

}
```

Consumer

```
While (1) {

while (buffer is empty);

remove an item;

consume the item

}
```


Often times, we have to wait for shared resources

- Busy waiting is a bad idea
- Checking resources itself needs to be in critical section
- Busying waiting inside CS even worse!
  - No one else can check!

→ Need a more powerful sync. primitive!

Reading assignment

- Chapters 3, 5
- Chapter 6 (process synchronization)
  - Read up materials covered