PURDUE UNIVERSITY®

CS 25200: Systems Programming

Lecture 22: Resource Allocation Graphs, Dining Philosophers, and Semaphore Review

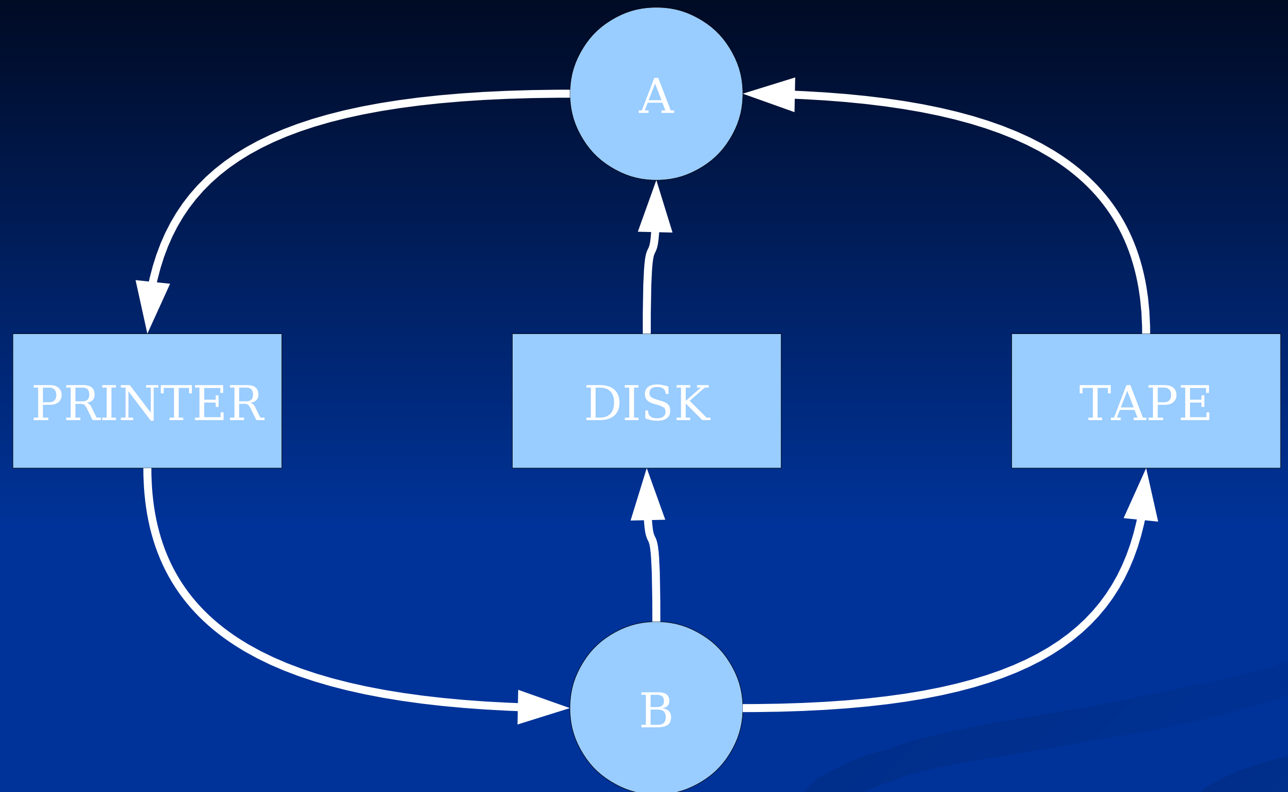
Prof. Turkstra

Lecture 22

- Resource allocation graph
- Deadlock
- Dining philosophers
- Semaphores

Resource allocation graph

- Circles represent processes
- Squares represent resources
- Assignment edge A ← R indicates that A holds resource R
- Request edge B → S indicates B is requesting S
- Cycle mean



- Program copies file from tape to disk, prints to printer
- A holds tape, disk requests printer
- B holds printer, requests tape and disk

Conditions for Deadlock

- Mutual exclusion – resource only assigned to exactly one process
- Hold and wait – multiple independent requests
- No preemption – resources can only be released voluntarily by the holder
- Circular chain of requests
- All four most hold for deadlock to be possible

Handling Deadlock

- Four strategies:
  - Detect and recover
  - Dynamic avoidance (careful allocation)
  - Prevention – eliminate one of the four previous conditions
  - Do nothing – pretend it doesn’t happen
    - Most OSs do this

Simple example

```
int balance1 = 100;
int balance2 = 20;
pthread_mutex_t m1;
pthread_mutex_t m2;
```

```
transfer1_to_2(int amount) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  balance1 -= amount;
  balance2 += amount;
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
transfer2_to_1(int amount) {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  balance1 -= amount;
  balance2 += amount;
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Preventing circular wait

- Ordering of resources
- Always request resources in ascending order
- Release in descending order
- Example
  - Tape: 0
  - Disk: 1
  - Printer: 2

Dealing with Deadlock

- Kill it.
- gdb might help

```
gdb> info thread              // list all threads
gdb> thread <number>          // switch to thread
gdb> bt                       // stack trace
```

Deadlock

- Deadlocks are often nondeterministic
- May have to run a program for a long time, or many times

Starvation

- Deadlock’s slightly less evil cousin
- Thread may wait for a long time before resource becomes available
- Eventually gets into the critical section, though
- Why mutexes use queues

Dining philosophers

- Deadlock
  - Two or more threads are blocked forever
- Starvation
  - One or more threads is unable to gain access to a shared resource and therefore unable to make progress
- Livelock
  - Two or more threads are caught solely responding to each other. No progress made, but they continue executing

Semaphore

- Synchronization variable that takes on positive integer values
  - Dijkstra, 60s
- Two operations:

  P(semaphore): atomic operation waits for semaphore > 0, then decrements by one
  - “Proberen” in Dutch
- V(sempahore): atomic operation increments by one
  - “Verhogen”

Implementation

```
typedef struct {
  int count;
  queue q;
} semaphore;
```

sem_wait (atomic)

```
void sem_wait(semaphore s) {
  if (s->count > 0) {
    s->count--;
    return;
  }
  add(s->q, current_thread);
  sleep(); // re-dispatch
  return;
}
```

sem_post (atomic)

```
void sem_post(semaphore s) {
  if (is_empty(s->q)) {
    s->count++;
  } else {
    thread = remove_first(s->q);
    wakeup(thread); // put thread on ready q
  }
  return;
}
```

Atomicity

- Remember, the previous definitions rely on hardware support
  - Disable interrupts on a uniprocessor system
  - Spinlock on multiprocessor
    - Atomic instruction(s)
- Left out for simplicity

POSIX Semaphores

- Declaration

```
#include <semaphore.h>
sem_t sem;
```

- Initialization

```
sem_init(sem_t *sem, int pshared, int value);
```

- Decrement

```
sem_wait(sem_t *sem);
```

- Vincrement

```
sem_post(sem_t *sem);
```

Semaphore

- count = 1 → mutex or lock
- count > 1 → permit n processes access
- count = 0 → wait for an event

Questions?